A Sheet Music Organisation System

Interim Report

Submitted for the BSc in Computer Science with Industrial Experience

January 2015

by

Charlotte Godley

Contents

1 Introduction

Notation of western classical music has used a combination of the diastematic and phonetic notation (**RRastall**) since the advent of Gregorian chant around 640AD (**RTaruskin**). The function of this notation falls into two main divisions: the expression of relationship in sound frequency, and the expression of relationship in time, or measure (**oxHistory**).

The key element of this form of notation is the staff, a grouping of five horizontal lines. This is important as each line or space in the staff indicates a different sound pitch, a term meaning the relative "highness" or "lowness" of the sound (**classroom**).

This staff is divided by barlines, vertical lines delineating grouped units of sound and silence (formally referred to as notes and rests), which provides an indication of the unit's relationship in time by its juxtaposition to other groupings in the composition. These groupings are called measures or bars, with each bar having a variable maximum of notes and rests.

The representations of other parameters in staff notation are normally phonetic, such as indications p - meaning piano, or "quiet" - and pizz - meaning pizzicato, or "plucked" (**RRastall**). This mechanism is complex in nature, and has a large but finite set of symbols which control every element of the composition.

A musician will often organise pieces notated using this system, or scores as they are formally known, in a music cabinet or filing cabinet (**musicOrganising**). These range from basic shelving to the more traditional large ornate cabinets where drawers only allow the user to see the top document, making it difficult to find scores which are deeper in the cabinet. This can often mean that music a performer will use regularly is kept on music stands and in piano stools, because if they were to return an item back into the music cabinet, it is likely that they would never find it again (**SheetMusicRant**).

This makes it difficult for instrumentalists to find a good way to organise their collections - this particular physical method allows for only one or two ordering choices, whereas the described notation system has many ways in which a piece can be identified.

It would be useful to a variety of musicians to be able to organise and search their collections using more than one mechanism. Such mechanisms have been developed in software automatically to cater for song title and composer name, but only allow for more detailed organisation by asking the user to provide more complex meta information (**calypso**). In a physical system this type of organisation would require file duplication. This project aims to create a sheet music organisation system for virtual music organisation, which will provide an automatic mechanism for the described problem.

This document discusses the aims and objectives of this project, background setting the project in a technical perspective, designs for the development of the solution, and a review of progress and project management to date.

2 Aims and Objectives

2.1 Project Aim

The overall aim of the project is to design and develop a sheet music library application, with the ability to organise and view personal sheet music collections, and download sheet music from the internet. Time permitting, it should also be able to generate sound from the sheet music, and import editable music from flat images.

2.2 Primary Objectives

The project will be considered complete if the following objectives are met:

2.2.1 Rendering of musical files

It is necessary that the project have the ability to render music files, as the intended solution is for the displaying and browsing of sheet music. The chosen file format is MusicXML, a form of XML which is standardised and used by many existing musical composition software solutions.

2.2.2 Extraction of Metadata

The project must be able to extract relevant and useful information about the pieces in the user's collection, as the project's aim is enabling automated collection organisation.

2.2.3 Connection to Online Music Collections

Further to browsing a user's personal collection, the project should enable users to search online music collections. This provides users with a better search mechanism than using a search engine, as it allows users to browse using technical terminology.

2.2.4 Develop the project using Test Driven Development

The project should be developed using Test Driven Development, as due to the complexity and amount of symbols required for music production, it is important that each feature and symbol be tested meticulously in order to ensure validity.

2.3 Secondary Objectives

The following objectives are to be completed if the primary objectives are met, and as such do not dictate the success or failure of the project, but rather are features which would add value to the project.

2.3.1 Audio playback

A further useful, but not mandatory feature, would be the ability to select and play parts of music files, enabling the automatic creation of accompaniment parts for solo musicians, amongst other benefits.

2.3.2 MusicOCR conversion of images to parseable MusicXML

It would be easier for musicians to merge their physical and virtual music collections for automatic organisation if the solution provided the ability to import flat image files and converted them to musicXML, using musical Optical Character Recognition.

3 Background

3.1 Problem Context

This problem's main focus is the difficulty of organising classical sheet music, and how this can be made easier by the automatic extraction of key pieces of information. In order to understand what a performer may want to know about a particular piece, it is important to have a brief understanding of the elements of musical notation common to all compositions.

3.1.1 Clefs

As mentioned in the introduction, an important part of musical notation is a sound's frequency relation, denoted by the staff lines and spaces.

In this system, sound frequencies, or pitches, are denoted by letters A-G - each set of these eight letters is an octave, after which the next pitch above it will be the start of a new octave.

In order to provide a link between the lines and spaces of a staff and pitch name, a clef symbol is necessary:

Each clef symbol denotes a different pitch name - in the above example, a G. The center around which this symbol is drawn - here, the second line from the bottom of the staff - indicates that this line or space will be known as a G. From this the reader can infer all other pitches by counting through the letters of the cyclic octave system, so in the given example, the pitch above becomes an A, and the pitch below becomes an F.

This symbol is important to a musician as different clefs are used to position the majority of the pitches in a piece on the staff, as this makes it easier to read. From this a performer can infer the average range of a piece, and predict whether this will be comfortable for the performer's chosen instrument or voice.

3.1.2 Keys

A second important indication to the player is the key, denoted by a key signature:

A collection of symbols at the beginning of the piece indicate which pitches should be raised by half pitches, and which should be lowered. Raised pitches are called sharps, indicated by the # symbol, whilst lowered pitches are called flats, indicated by the [flat] symbol. Each key, which has a letter name and key type, has a different combination of flats or sharps.

This is a useful piece of notation to a musician as pieces in less common keys, such as C# major or F# major, may prove more difficult for the user to perform, and therefore they may want to filter out pieces in these particular keys. Similarly, in the case of singers, a singer's range may sit comfortably in one or two keys and they would perhaps want to find pieces in only these keys.

3.1.3 Meter

The third symbol denoted at the beginning of a measure is the meter, two numerals positioned like a mathematical fraction:

The most common meter is 4/4, sometimes denoted by a C indicating "Common time". The upper number of a meter symbol indicates the amount of beats in the bar. A beat simply refers to a note or rest, and the type of beat is indicated by the lower number. In this case, 4/4 indicates a measure will contain 4 crotchets, or quarter-length notes.

This information is important as it tells the performer how the rhythm and beat of the piece should be felt, counted and performed, and is useful for searching purposes as different meters, or time signatures as they are sometimes referred to, give the piece a different feeling, dictating the sort of occasion this piece would accompany.

For example, 2/4 is commonly used for march pieces, 3/4 is commonly used for waltzes and dance pieces, and 6/8 gives a similar, but more syncopated feel of a dance like piece.

3.1.4 Tempo

The speed of a particular piece, or the tempo, is indicated by an equation:

As explained above, this equation shows that the piece should be played at 60 beats per minute - the symbol dictating the sort of beat per minute depends on the time signature, here a crotchet (or quarter note) is given as the piece is in 4/4 time. Sometimes, this will be accompanied by a text direction to indicate speed or style, such as Andante, indicating a walking speed.

This indication would prove a useful identifier as pieces of different tempos provide variation in performance lists, so a concert organiser may want to find pieces with a variety of tempos.

3.1.5 Further metadata

Aside from these symbols, there are some items of textual information useful to the user.

The first of these would be the parts in the piece and their transpositions. Parts would be relevant as a particular group of instrumentalists may need parts that fit their instruments. If this is not the case for a given piece, however, a part written for a different instrument, for example, the Alto Saxophone rather than the Tenor Horn, may be compatible with the instrument anyway, if the transposition matches the instruments together.

Further to this, the user would want to know the piece's title, and names of publishers, composers, arrangers and lyricists of the work. Further to the composer name, it may be useful to know the date of composition as an indication of the piece's stylistic era, such as Classical/Baroque/Romantic, though this would not always be written on the sheet music so may need to be researched using the internet.

3.2 Comparison of Technologies

3.2.1 Programming Language

This project could be developed with a variety of programming languages, as displayed in the following table:

Language	Speed of development	Developer's Knowledge	Most recently used	Cross compatible
C#	Fast	A lot	2nd year	Yes
Python	Fast	A lot	In constant use for over a year	Yes
C++	Slow	Average	2nd year	Yes

The four key elements the developer is focussing on are speed of development, as the time constraint of a year means it is important that development is not hindered by the language itself, developer knowledge and most recent usage of the language as this will provide an additional time benefit, and cross compatibility, due to the different operating systems the developer intends to use in the course of development.

The developer understands C# is cross compatible through the use of the Mono Project, which is feature complete to C# 10(**MonoDev**), or Xamarin Studio and other such tools, but has not developed any applications

with c# for multiple operating systems. For this reason, the developer feels more comfortable using Python, owing to the experience of writing applications for Linux and Windows in previous projects.

Due to these factors, Python has been selected, particularly as the developer used much of the features of python during an industrial placement and feels this is the most appropriate language for rapid development.

Further to these benefits, there are many projects in the field of musical software research currently in existence using this language, (**pmus**) which will help when trying to debug issues and build upon previous research.

As a further technical challenge, the developer has elected to use Python 3, as this is the most up to date version, but also a version the developer has yet to work with and may requre some relearning of specific areas of syntax. This presents a technical challenge as many other open source projects and packages use version 2.7, so interfacing with other works may require submitting pull requests to update them to version 3.

3.2.2 File format

The project will require at least one default format for it to process music, which needs to have detailed information about what the score contains. The table below describes the options considered by the developer:

3.2.2.1 Creating a new file format It would be possible for this project to create it's own method of file storage, similar to methods used by commercial and open source composition software. This would enable the developer to build a format from the ground up, and design it around the way the system would work.

However, this project focusses on displaying and organising sheet music, and will not be allowing the user to create new sheet music from inside the program. Therefore, in order for the project to be a success it is important that the file format be compatible with popular composition software packages.

3.2.2.2 Using a previously created file format It would potentially be possible to examine files generated from popular packages such as Sibelius, the world's best-selling music composition software, (**avid**) or MuseScore, an open source offering to the composition industry. This would mean that the program would be directly compatible with the default files created by each package.

However, this would require further research into how each piece of software generates it's files, and in the case of Sibelius, may incur copyright issues due to Sibelius being commercial software. It would also mean the project would be tightly coupled with that file format, and if the developers of the original file format were to change it in future, modifications would have to be made frequently to the file handler.

3.2.2.3 Using a well documented music format The final choice considered is using a well documented music format referred to as MusicXML. This format was designed from the ground up to allow sharing of sheet music between programs, and in order to archive sheet music for the future (**mxml**).

This file format is directly compatible with MuseScore(**MuseTour**), and compatible with Sibelius version 7, and earlier versions through the use of a plug in (**Plugin**). The MusicXML website provides a well documented tutorial on producing musicXML as well as a support forum and list of all tags available in musicXML.

However, using this file format means that some of the decisions on how to organise sheet music would have been made according another developer's wishes, which cannot be fixed or improved upon as would be possible if the developer chose to create their own file format.

The problem with tight coupling to this format may also be incurred, however, as this format is intended for sharing between programs, it is unlikely issues of backwards compatibility will occur.

3.3 Comparison of Algorithms for Rendering and Organising Sheet Music

3.3.1 XML parsing algorithms

3.3.1.1 DOM loading algorithm The first option is using a **DOM library** - in python, there are two built in libraries, called DOM (Document Object Model) and MiniDOM, a cut down version of the first. In this method, the entire XML file is loaded into memory, and the developer can look for specific tags and data using search functions.

This option has not been chosen for either objective. This is because for the purpose of rendering music, it may not be necessary to load all of the formatting information from the file, as well as some of the encoding data which composition software often puts into the file - this means that loading all of the data into memory is unnecessary.

Secondly, the DOM library in python is not very easy to use and having to search for a specific tag name does not make for rapid development. Thirdly, in reference to the searching and organising portion of the project, selection of metadata should not require a whole file of data, but rather select tags such as the instruments in the piece, composer, key, tempo and other such information.

3.3.1.2 SAX parsing algorithm SAX is a Simple API for XML processing, which parses the tags in the XML one by one, connecting to callbacks when specific things occur in XML parsing. This enables the developer to build up the functionality of object loading gradually, by connecting specific found tags to created handler methods, and allows for ignorance of tags which are not required.

Furthermore, in the area of metadata extraction, it is unlikely the process will require the entire file in order to extract key features of the piece, and therefore SAX parsing is far more suited to both tasks.

A further musical benefit to loading and rendering sheet music is that this method of parsing could enable the program to disregard notation considered to complex for the user to understand, for example when teaching a new student music theory.

3.3.2 XML verification algorithms

For both the algorithm options discussed in the above section, a further choice is whether to verify the XML parsed, using an online file validator, or presume the file is written in valid MusicXML.

The usual choice is to verify all XML, and is therefore the default option for both methods of parsing. Whilst this confirms that XML is valid before starting parsing of a file which could be corrupt, the speed at which files will parse is greatly reduced according to the speed of the user's internet connection. Furthermore, if the user is browsing their own music collection, it should not be necessary for the user to be connected to the internet.

Due to speed and functionality considerations, the choice has been made that the XML parser algorithm will not verify XML being converted to objects, or being examined for metadata. Given that most musicXML will be produced automatically by other programs, it is unlikely files opened by the project will be corrupt, though necessary steps will be taken to avoid this causing a problem in the program.

3.3.3 Loading and memory management algorithm

This project will load musicXML files for rendering into memory using a class structure, with each class providing it's own interface to output mechanisms. This has been decided as objects provide clean and navigatable structure, with the ability to inherit or overwrite parent class mechanisms.

It would also be possible to extract metadata from the musicXML files and create a converter directly to the output format for rendering, however this would make the structure harder to navigate and more difficult to debug, and would probably result in bad programming practices being used.

3.3.4 Metadata algorithm

The metadata algorithm has been designed so that, for a given folder, the program will parse all of the files with the XML extension for a given selection of information (for example, composer, piece title, instruments) and store this information in memory.

This is to be indexed either by the information title - e.g "composer"; or by the information itself - e.g "bartok". This will facilitate faster searching of the database for use when the user is finding a particular piece, and facilitate the production of auto generated playlists by the system. Depending on the method of indexing, the alternate indexer should be stored as part of the value in a key value pair format, alongside the file in which it was found.

There is a choice to be made as to how often this algorithm should be run, as discussed in the following sections.

3.3.4.1 Running the algorithm on every application open It would be possible to check and load all of the metadata for a particular folder each time the user opens the application. This would ensure if any file changes have been made that the metadata was up to date.

However, with testing, it has been found that this is a slow method owing to the volume of test data, and will only become slower if a user has a large collection of music.

3.3.4.2 Caching previous runs It has been decided that the metadata algorithm will first check for a cached metadata file, created from a previous run of metadata parsing. It will then look for any files not in this cached file, parse the metadata from each file, and save out to the updated cache.

This avoids repeat loading of the same data, but if files are changed or updated after the first application open, it could cause confusion for the user. It may be possible to provide an option in the program to force the parser to re-run the metadata extraction on all files in the folder if this is an issue.

It has been decided that the file format for caching metadata will be SQLite or similar light database implementations, as this will avoid the outputted data being tightly coupled with the program, and therefore will make the architecture more extendable in future developments. Another option considered for storage is simply serialising the chosen data structure, which would be quicker to develop. This has been decided against as this would couple data output with the solution.

In order to store this information in a data structure, the following methods have been considered.

3.3.4.3 Using object oriented organisation It would be possible to store the information in a class structure, with each class holding the selected information and the file in which the metadata was found.

However, the described algorithm will only be storing 3 elements of data, all of which are strings, and therefore will not need a complex structure.

3.3.4.4 Using generics A further option would be to use the python built in type of a dictionary. This would make more sense than an object as there are only 3 key items of information to store, and allows for indexing using different mechanisms. This option has been chosen, in combination with the described type of data storage, in order to aid rapid development and provide a simple program structure.s

3.3.5 Rendering Algorithm

The program is required to take the object structure and transform it, in some way, to musician readable sheet music.

3.3.5.1 Creating a new algorithm using fonts It would be possible to create sheet music using an algorithm designed by the developer. A potential option would be to create an automatic method of typesetting every class using music fonts layered on top of each other inside the render window.

This would give complexity and challenge to the developer and allow the developer to tailor optimisations according to speed and memory management.

However, this may create problems such as panning and zooming into the music which could prove complicated, how best to layer fonts on top of each other which may be difficult to do in a graphical window, and debugging the process would be difficult as it would require manually visually checking the algorithm functions and generates the correct sheet music.

Furthermore, music software has existed for a long time and the rendering algorithm will have been covered by many developers, who will have had longer to develop and test the solution, therefore developing a new algorithm may be considered reinventing the wheel.

3.3.5.2 Creating a new algorithm using images It may also be possible to apply the same algorithm, but generate an image which would then be displayed using a graphical viewer from a built in python graphics library, which would have the functionality for panning and zooming around the image built in.

This still, however, leaves the problem of debugging the process manually, and the described issue with reinventing the wheel.

3.3.5.3 Outputting to a separate rendering program A third option is to have the program connect each class with a separate program, developed by a third party which will render the given symbols.

This removes a level of complexity and technical challenge, but avoids covering a research area which has already been done. Furthermore, this project's aim and main focus is to organise and import music, rather than render or create it, and recreating a rendering algorithm would not achieve this goal.

In addition to this, the issue with debugging music rendering would be alleviated as a third party program designed to render music would have been tested extensively, so the program could be debugged by simply ensuring outputs to the program are correct.

It has been decided to use this option and use Lilypond as the third party software. Lilypond is a cross-platform typesetting language, devoted to producing the highest-quality sheet music possible (**Lilypond**). It has a large body of users and a well written collection of documentation.

This will add a further technical challenge as it will be required to understand the syntax of Lilypond, and how Python should output Lilypond files for compilation and conversion to PDF.

3.4 Comparison of Technologies for Importing Online Musical Sources

3.4.1 Musical Sources

Two open and free sources of sheet music have been selected for potential inclusion, which will enable users to connect their own music collections with new music without using a browser to peruse collections. The first is **MuseScore Online**, which is a community website created for composers to upload share and discover compositions using the MuseScore platform (**MuseShare**).

This has been selected due to the number of files available, the openness of the platform and the well documented API created by the developers. It will, however, be necessary to manage copyright issues, as pieces published on this website may be published under the license of the composer's choosing and therefore may cause issues with certain types of users, in particular those performing commercially.

The second selected source is the **IMSLP**. This is the **International Music Score Library Project**, built with the intention of sharing the world's public domain music and contains 290,000 scores to date (**imslp**). This may be a questionable source, as not all pieces are uploaded in MusicXML format due to the pieces being scanned and uploaded by community members, rather than being automatically generated by a piece of software. However, this source does not raise any copyright issues as all pieces are no longer covered by copyright.

It may also be possible to import collections from subscription services and websites enabling purchase of music, such as **MusicNotes.com**. However, this will require closer contact with the companies maintaining the website and may not be appropriate for an educational and academic purpose.

3.4.2 Searching Algorithm

The APIs for both selected sources provide a variety of output formats, the 2 most prominent being XML and JSON. It would be possible to use an algorithm which repeatedly connects to the API and polls for the relevant input from the user, returning a list of options which the user would then select from and download from the server. However, from the perspective of the user, this would be slow, requiring repeated connection to the internet. This would also cause problems for the maintainers of the server, as repeated requests from a piece of software would cause a heavy load on the server and be very unnecessary.

It has therefore been decided that the software will cache a copy of all metadata served from each online source, and search for the relevant inputted data from this, and then, if necessary, collect the relevant file from the server. This would require a connection to the server only twice - once when updating metadata sources, and once when downloading a file - rather than a persistent or repeated connection.

3.5 Comparison of Algorithms for Sound Output and Image Input

3.5.1 MIDI algorithm

The sound output algorithm must, for a given part or selection of parts, output the sheet music to a MIDI or MP3 file, which can then be played within the program.

It has been decided that each class in the solution will have a method to produce this output. Each class holding objects as members will call each sub object's sound output method, which will be saved by the program as a full outputted file and played using a python sound output library.

This provides an extendable architecture, as the developer could choose to produce different outputs, such as one for MIDI and one for MP3, which would be called in the same way.

3.5.2 Image input algorithm

In order to import images or flat files into the chosen file format, it will be necessary for the program to include the ability to apply music optical character recognition to the file, and save the output to MusicXML, which can then be parsed by other parts of the program.

3.5.2.1 Creating a new algorithm

It would be possible for the developer to produce their own algorithm for converting new imported images into the chosen file format. This would allow the developer to optimise the algorithm according to their own wishes, and provide sufficient technical challenge.

However, this project is concerned with music organisation, not OCR specifically, and as such the project is too large to commit a sufficient amount of time to this particular algorithm in order to make it function as well as other algorithms.

As a reference point, Optical Character Recognition for natural languages has taken may years to develop and perfect, and has been an attractive research area and idea to a wide variety of users(**InternationalConf**). OMR, or Optical Music Recognition, has been the focus of international research for over three decades, and while numerous achievements have been made, there are still many challenges to be faced before it reaches its full potential(**musicocr**).

3.5.2.2 Using third party software

It has therefore been decided that OCR as a topic is too large for this project, and if this goal is included in the project, it will be through communication with other systems, such as Audiveris, an open music scanner (**audiveris**).

This removes the technical challenge of producing an entirely new algorithm, but adds the challenge of understanding how optical music character recognition scanners work, and how they can be integrated with the system, particularly if the third party package is not developed in Python.

3.6 Alternative Solutions

The majority of music software available to the masses covers the creative process of writing music, with the two most well known packages being the aforementioned Sibelius, a commercial software package costing around £500 (**avid**), and the freely available and Open Source MuseScore(**MuseTour**).

4 Designs

4.1 System Design

4.1.1 Class diagrams and mind map

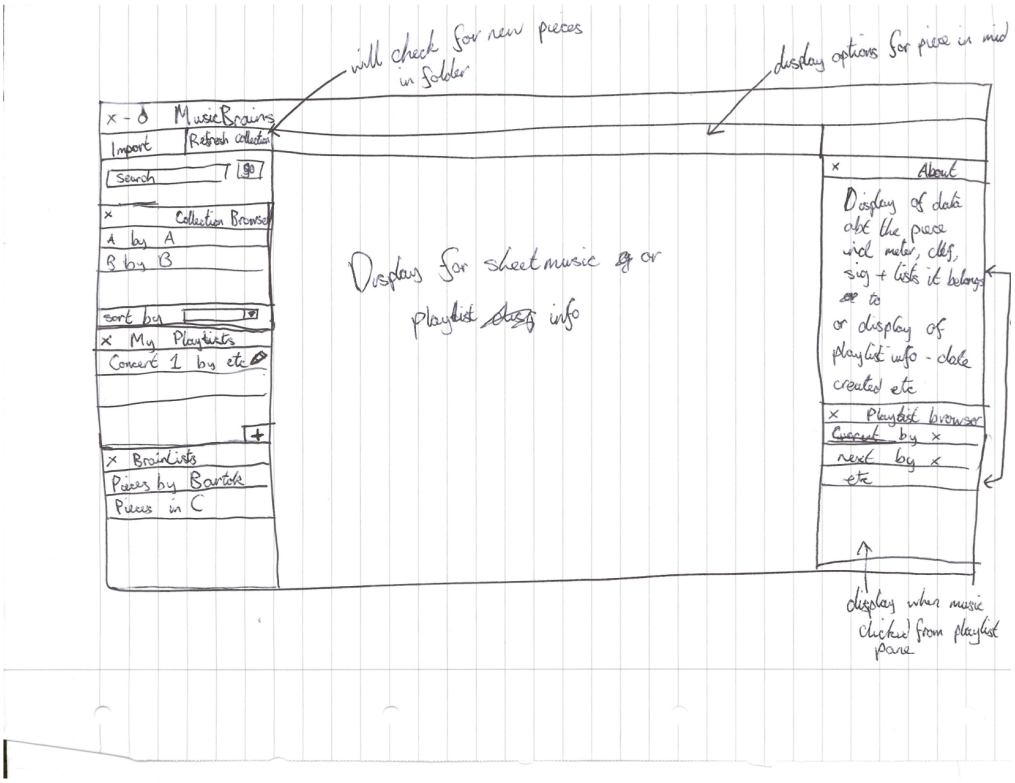
The developer used a mind map to initially appraise the connection between each musical symbol. This helped break down a piece from a musician's view, and showed what information would be necessary between each symbol's class.

An initial class diagram was drawn from the mind map. This was modified during the course of development testing and research of the initial model. In particular, the developer looked at other sources such as Music21, a toolkit for computer-aided musicology(**Music21**), which helped the developer to examine whether the initial model was missing any classes or attributes. The diagrams described are in the appendices.

4.2 UI Design

The User interface for this project is designed with the average musician in mind, and the interfaces that user would normally have used. With this in mind, the designs explained below and shown in Appendix D are inspired by other music applications in common usage.

4.2.1 Main Display



The above image shows the main view of the application. Various panes to the left can be closed using the X button and show different ways the music can be displayed, either as individual units or as playlists. The panes to the right display content based on what is selected for display in the middle, larger window, generally

either a playlist or a singular piece of sheet music. Updates to these panes and various pop up boxes associated with different buttons in the display are in the appendices.

4.2.2 Musician feedback survey

In order to understand how well this user interface works with a variety of users, a survey was designed which will be given to a selection of musicians, who will feedback on how easy the UI is to use and any updates which should be made to improve it. This feedback session will be performed after the initial sketches are made into a virtual user interface with no back end connected to the buttons. An example survey is provided in the appendices.

4.3 Test Design

The developer has chosen to use test driven development. This is an agile software development methodology which utilises the rules that the developer should never write a signle line of code unless there is a failing automated test(**TDD**). This methodology has been chosen as the nature of the notation of music means that meticulous detail must be payed to how and with what symbol every element is notated, and Test Driven Development will ensure that every detail has been validated and proven to work.

The list of tests to date, which are written on an adhoc basis as the developer builds up functionality, are attached in the appendices, along with a description of current testcases used to confirm the functionality works with real world examples of sheet music.

5 Project Management Review

5.1 Current progress

- body of research done and decisions made from this - development of code and designs for user interface decisions - compare this with expected result

5.2 Adjustments made

based on coursework deadlines issues with stress/multiple projects being handled review and modifications made, and future things to consider in project management based on these

5.3 Revised timeplan

decision on objective implementation: OCR

Appendices

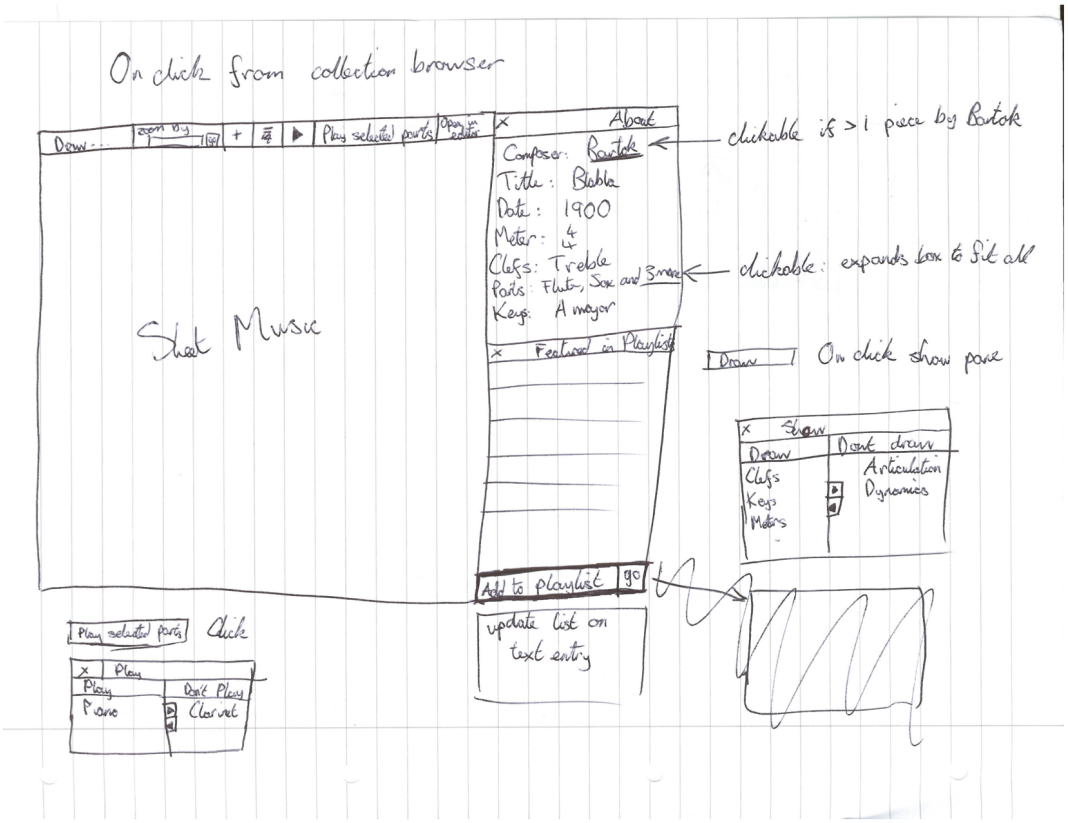
A Mind map of elements of Music

B Initial class diagram

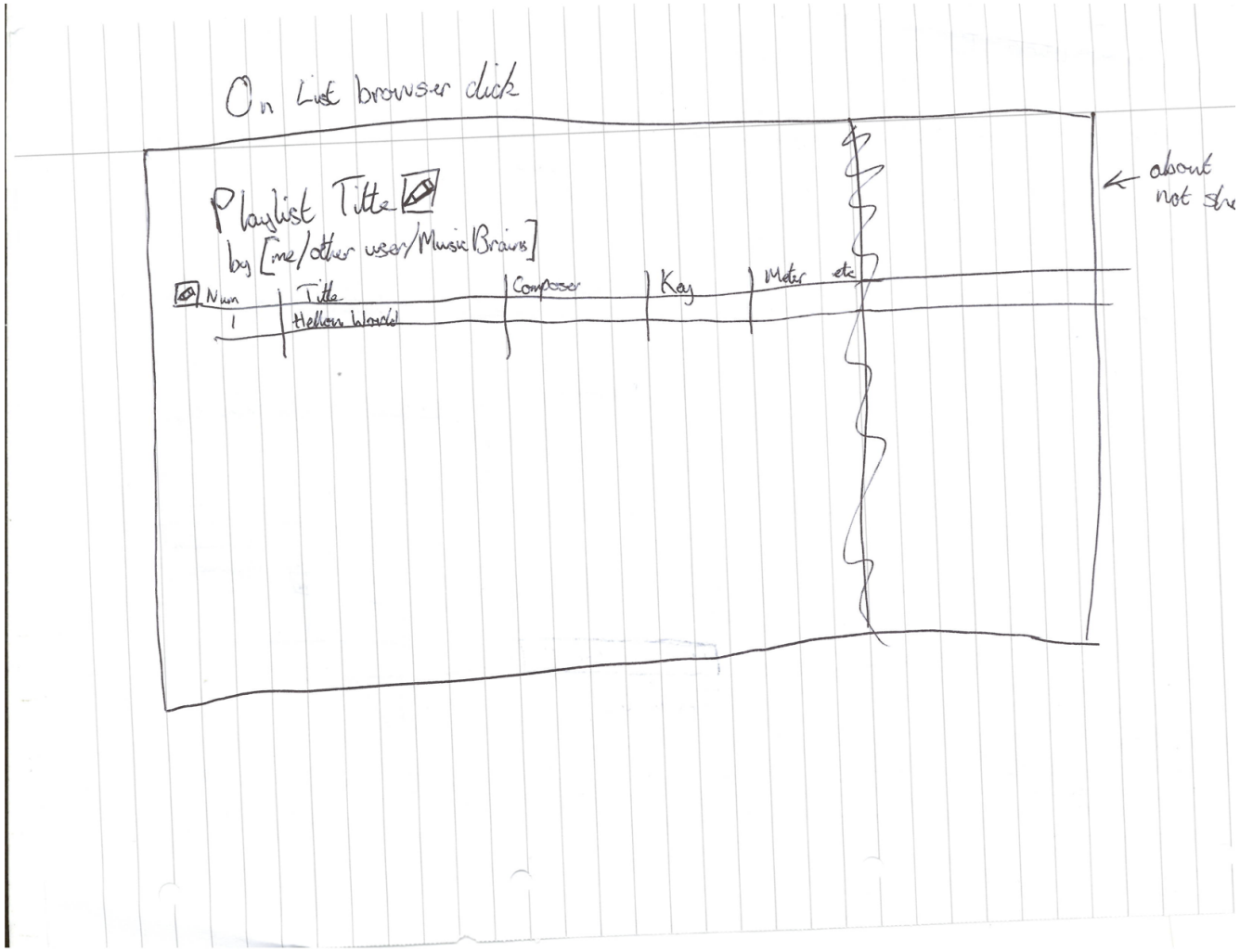
C Revised computerised class diagram

D Initial User Interface Design

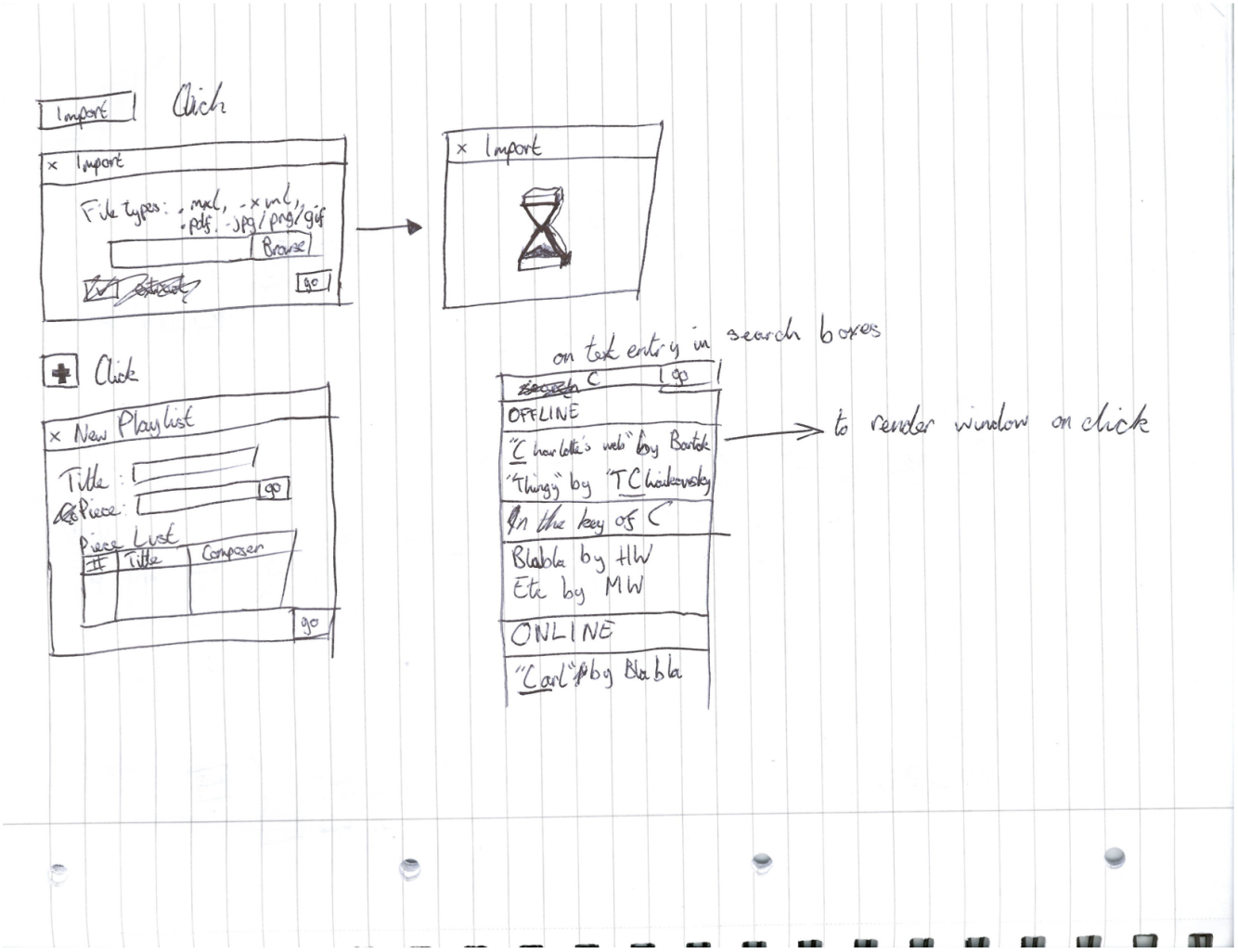
D.1 Changes to main display when sheet music selected



D.2 Changes to main display when playlist is selected



D.3 Other Panes and Popup windows



E User Interface Feedback survey

F Revised User Interface Design

6 References